Algorithm Design and Analysis

Case Study of Dynamic Programming Approach Vs Greedy Algorithm Approach

Introduction:

A divide and conquer algorithm partitions a problem into sub-problems, solves the sub-problems recursively and then combines their solutions to solve the original problem. But there are instances where the sub-problems share sub-sub-problems, in such scenarios a divide and conquer does more work than necessary, solving the common sub-problems repeatedly. In such scenarios, we resort to solving these problems using other alternative algorithms which also solve the problem by solving the sub-problems but with a variation.

Dynamic Programming: Dynamic Programming also solves the problem by combining the solutions to the overlapping sub-problems. But dynamic programming solves each sub-problem only once, thereby reducing the number of computations and then keeps a record of the solution for each sub-problem. Next time when a previously computed result is needed it is simply looked up through a O(1) complexity table.

Greedy Algorithm Approach: A greedy algorithm solves the problem by making the locally optimal choice at each stage with hope of finding a global optimal. A greedy algorithm makes a choice that seems best and then solves the sub-problems that arise next. The choice made by greedy algorithm may depend on the choices made previously but doesn't depend on the future choices or all solutions to the sub-problems like in case of Dynamic Programming.

The main difference between greedy algorithms and dynamic programming is that instead of first finding optimal solutions to sub problems and then making informed choice, greedy algorithms first make a greedy choice, the choice that looks best at the time, and then solves a resulting sub problem, without bothering to solve all possible related smaller sub problems.

Proposal:

In this project, we intend to apply both these algorithms on a set 6 problems statements and then crucially compare the performance of each algorithm for each problem statement. We will further conclude the cases in which one of these algorithms will be a better choice.

Parameters for comparing the algorithms:

- We will compare the algorithms by calculating the space (Γ) and time complexity (Θ) in each case. Graphs for the computations will be drawn accordingly where needed.
- We will also compare the algorithms based on the optimal solution provided for each problem statement.

Based on the above parameters we will determine which algorithm performs better for a specific problem.

Case Study 1 - Coin change problem:

Problem Statement:

To find the smallest number of coins required to make the exact change of an amount in a given monetary system.
Example:
Coins Denomination = {1, 5, 10, 25}
Amount = 35
Answer = 2 {35 = 25 + 10}

Dynamic Programming Approach:

Characterize the structure of an optimal solution:
The coin change problem exhibits optimal substructure. It can be observed as follows:
Let's assume **S** be the optimal solution to make change for **a** amount using coin denominations of $d_1, d_2, \ldots, d_k$.
Remove arbitrary denomination d_k from S and call it S'.
$S' = S - d_k$
If S is optimal then S' must be optimal for $a - d_k$ amount.
To prove this, let's assume there is a better solution other than S' to make change of $a- d_k$ amount and it is called X. Replace S' with this better solution X. This means if d_k is added to X, it will result in a solution with fewer number of coins than the initial solution S. This contradicts the original assumption that S is optimal. Therefore, it concludes that if S is optimal S' will also be optimal.
This shows that the coin change problem comprises of optimal solutions to smaller sub problems and an optimal substructure exists.

Recursively define the value of an optimal solution:
Let Coin[p] be the minimum number of coins of denominations $d_1, d_2, \ldots, d_k$ required to make change of p amount. Take any first coin d_i, where $d_i <= p$. Because the problem exhibits optimal substructure, if we remove d_i the remaining coins in Coin[p] must be the optimal solution to make change of $p- d_i$ amount. As d_i is the first coin to come up with change for p amount this means that

$$Coin[p] = 1 + Coin[p - d_i]$$

In other words, take one d_i coin and coins to make change for $p- d_i$ amount to get total coins for making change for p amount. As d_i is not known so all k possibilities for d_i is checked. Also, the base condition that 0 coins are needed to make a change for 0 amount holds. The recurrence obtained is as follows:

$$Coin[p] = 0 \qquad \text{if } p = 0,$$
$$Coin[p] = \min_{0 \le i \le k}\{1 + Coin[p - d_i] \qquad \text{if } p > 0.$$

Compute the Value of the Optimal Solution:
For the solution
d = array containing denomination values
n = amount to be changed.

```
COINCHANGE(d, n)
1 Coin[0] = 0
2 k = len(d)
3 for p in 1 to n
4       min = ∞
5       for i in 1 to k
6               if d[i] ≤ p
7                       if 1 + Coin[p – d[i]] < min
8                               min = 1 + Coin[p – d[i]]
9                               lastcoin = i
10      Coin[p] = min
11      Solution[p] = lastcoin
12 return Coin and Solution
```

Running Time:
The running time for the above procedure will be $\theta(nk)$ because of the nested loops which run for 1 to n and for 1 to k. The solutions are stored in arrays Coin and Solution. The previously computed values are looked up from the Coin array and it does not contribute to the running time.

Greedy Approach:

Optimal substructure:
It's already shown in the Dynamic Programming approach that the Coin Change problem exhibits an optimal substructure.

Greedy Choice Property:
The objective of the problem is to find the minimum number of coins to make change of the given amount. If the largest denomination coin is chosen first, it will decrease the total number of coins. If the greedy choice is made and the largest denomination coin is chosen, the remaining sub-problem to solve would be select the least number of coins to make change of the remaining amount. It is already shown that the problem exhibits an optimal substructure so taking the largest denomination(d_1) first will leave the sub-problem S' and optimal solution to original problem will consist of coin d_1 and all the coins in an optimal solution to the sub-problem S'.
Thus, by making a local optimal choice a global optimal solution could be achieved.

Algorithm Pseudo Code:

```
COINCHANGEGREEDY(d, n)
1 Sort d in increasing order
2 Coin = empty
3 While n != 0
4       dk = largest denomination coin && n > dk
5       if dk = null
6               return "No solution"
7       else
```

8 $n = n - d_k$
9 $Coin = Coin \cup d_k$
10 return len(Coin)

Running Time:
The running time of the above procedure will depend on the sorting algorithm to sort the coin denominations in d. If k is the length of d then it could be done in O(k logk). The while loop will have the worst case running time of O(n). Therefore, the overall complexity of the procedure will be O(k logk) + O(n).

Comparison of DP vs Greedy:

Complexity	Dynamic Programming	Greedy Algorithm
Time	θ(nk)	O(k logk) + O(n)
Space	n	1

The greedy approach will give an optimal solution without having to maintain an extra space for storing the previous computations as in DP. Greedy approach will work for most practical currency denominations, but it might fail in certain scenarios and will give an incorrect solution. Eg: if the coin denominations available are {1, 5, 10, 20, 25} and the amount to be changed is 40. The greedy approach will give solution {25, 10, 5} but the optimal solution will be {20, 20}. A DP approach on the other hand is exhaustive and will always give the optimal solution irrespective of the coin denominations.
To conclude in most real-world scenarios greedy will be a better approach for the coin change problem, but it might fail for some of the denominations.

Case Study 2 - Increasing subsequence count:

Problem Statement:

Given an array of n positive integers A1, A2, …, An. Find a subsequence A[i,j] containing Ai, Ai+1, …, Aj, where 1<= i <= j <= n and subsequence A[i,j] is increasing, i.e Ai < Ai+1 for all i, 1<=i<N.
Example:
If A = {7, 3, 8, 4, 2, 6}, then the longest increasing subsequence will be {3, 4, 6}

Dynamic Programming Approach:

Characterize the structure of an optimal solution: For the longest increasing subsequence problem, assume that s is the optimal solution and it contains a_{j1}, a_{j2}, a_{j3}… a_{jk-1}, a_{jk}.

$$s = \{a_{j1}, a_{j2}, a_{j3}\ldots a_{jk-1}, a_{jk}\}$$

This means A[] does not contain an element after a_{jk} is greater than a_{jk}. And there doesn't exist an element with index between $j_{(k-1)}$ and j_k, that has a value between corresponding elements of subsequence s.
To prove this, remove a_{jk} from s and we call it s'.

$$s' = \{a_{j1}, a_{j2}, a_{j3}\ldots a_{jk-1}\}$$

If s is optimal then s' must be optimal for A - a_{jk}.
Assume s' is not optimal and there exists another optimal x. Therefore, if we add a_{jk} to x we get a better solution than s. This contradicts the assumption that s is optimal.
Therefore, the longest increasing subsequence problem exhibits an optimal substructure.

Recursively define the value of an optimal solution:
Let B(i) be the size of the longest subsequence starting from the i^{th} position.
B() will contain at least one element for n = 0 because the length of the longest subsequence will be at least one as the longest subsequence will at least have one element. For the remaining case, it can contain all numbers from i + 1 to n, if the number is greater than or equal to the last number. The recurrence could be obtained as follows:

B(n) = 1 if n = 1
B(n) = 1 + max{B(j), where i < j <= n and a[j] > a[i]} for 1 <= i < n

Compute the Value of the Optimal Solution:
a[i] = number at position i
B[] = array to save the size of the longest subsequence

```
LongIncSub(a)
1 n = len(a)
2 B[n] = 1
3 for i in n – 1 to 1
4         B[i] = 1
5         for j in i + 1 to n
6                 if a[j] > a[i] && 1 + B[j] > B[i]
7                         B[i] = 1 + B[j]
8 return B
```

Running Time: The running time for the above procedure will be $O(n^2)$ because of the two for loops running for n iterations.

Greedy Approach:

Optimal Substructure: As it has already been established before, the longest increasing subsequence exhibits an optimal substructure. This satisfies one of the two conditions for applying the greedy approach.

Greedy Choice Property: This problem doesn't exhibit a Greedy Choice property directly, it not possible to make a greedy optimal choice locally to end up with a global optimal solution. The longest increasing subsequence depends on the previously selected element to form the longest subsequence.

Patience Sorting: As explained before it is not possible to make greedy choice for the longest increasing subsequence problem. But this problem could be solved by doing a slight variation in the approach and using patience sorting. Patience Sorting is a one-person card game of solitaire, where a deck of cards labeled 1, 2 ,3, 4…n is taken. The deck is shuffled and cards are picked one at a time and arranged into piles on the table. The game follows the below rule:

- A low value card may be placed upon a higher value card or may be put into a new pile to the right of the existing piles.

For each step the top card on each pile is visible. If the next card picked is of higher value than the cards visible, then that card must be placed into a new pile to the right of the other piles. The objective of the game is to finish with the fewest piles possible.
This game follows a greedy strategy and it is to always place a card on the leftmost possible pile. Because of the greedy strategy at each step the value of the top cards are increasing from left to right.

The same principle of patience sorting can be applied to find the length of the longest increasing subsequence in the following manner. An array will n elements when sorted using patience sorting will result in l number of piles which will be the length of the longest increasing subsequence. If numbers $a_1 < a_2 < .. < a_n$ appear in increasing order then each number a_i must be placed in some pile to the right of the pile containing a_{i-1} number, because the number on top of that pile can only be less than the previous number in that pile. Also, when the greedy strategy is used if a number is placed in a pile other than the first pile, a pointer is added from that number to the current top number a' < a in the pile to the left. At the end of the procedure, let al be the number on the top of the right most pile, then the sequence:
$a_1, a_2,\ldots, a_{l-1}, a_l$ obtained by following the pointers form an increasing subsequence whose length is equal to the total number of piles obtained.

Sample Implementation Pseudo Code for Patience Sorting:

PATIENCESORT(n, A)
1 $S_1, S_2, S_3,..S_n$ = Empty Stack

```
2 For i in 1 to n
3       S_j = Find left most stack that fits A[i] //Binary Search could be used
4       PUSH(S_j, A[i])
5       if j = 1
6               predecessor(A[i]) = null
7       else
8               predecessor(A[i]) = PEEK(S_j-1)
9 return number of non-empty stacks
10 return sequence formed by following pointers from top number of rightmost non-empty
stack
```

Running Time:
The priority sorting procedure can be implemented to run in O(n logn) time complexity. An overhead of adding pointers is incurred in order to find the longest increasing subsequence.

Comparison of DP vs Greedy:

Complexity	Dynamic Programming	Patience Sorting(Greedy)
Time	$O(n^2)$	O(n logn)
Space	n	n

As seen in the case study above, it is difficult to directly implement Greedy approach to the longest increasing subsequence problem. The Dynamic Programming approach will always yield results with worst case running time of $O(n^2)$. The DP approach can be tweaked to perform better by better selection of data structures for implementation. When the greedy approach is applied using Patience sorting, even though it yield a solution of O(n logn), an overhead of maintaining the pointers to the previous smaller number is incurred. To conclude, using a DP will be a better approach for this problem.

Case Study 3 - Majority Element:

Problem Statement:

Given an array of size n, find the majority element. The majority element is the element that appears more than floor(n/2) times. Assume that the array is non-empty and the majority element always exist in the array.

Optimal Substructure: In the given problem assume that a is the majority element for a given sequence and remove any two different elements from the given sequence. It can be argued that the majority element a will also be the majority element in the new sequence (original sequence – two different elements). Thus, the problem exhibits optimal substructure.
The majority element could be found by finding the majority element in the smaller sequences and then combining the results.

Dynamic Programming cannot be implemented:
Though this problem exhibits overlapping sub-problems a direct dynamic programming approach cannot be applied as it is not necessarily to retrieve the results of the previously computed sub problem. In other words, we cannot apply memoization. Even if the results of the previously computed sub-problems are stored, it will not be a pure dynamic programming solution.

The problem could be approached using a hash map which will store the count of all the elements as follows:

```
MAJORITY(A)
1 hashmap = {}
2 for i in 0 to len(A)
3       element = A[i]
4       if hashmap contains element
5               hashmap[element] = hashmap[element] + 1
6       else
7               hashmap[element] = 1
8       if hashmap[element] > len(A)/2
9               return element
```

The above procedure will scan the array once and will calculate the count only once, thereby yielding results in O(n) time, but the procedure will also require O(n) space to store the values of the count of each element.

Greedy Algorithm cannot be applied:
Again, the problem cannot be solved using a greedy approach as there is no greedy choice property. We cannot make a greedy choice locally to pick an element and hope it to be the majority element in the entire array.

Another approach to solve the majority element problem is by using Moore's Voting Algorithm, which works on the principle that if each occurrence of an element a is canceled

with other elements that are different from a then, a will exist even in the end if it is a majority element. Using this algorithm, it can be determined if there exists a majority element and then a comparison can be made to check if it is greater than n/2, where n is the length of the array. The pseudo code for this implementation is as follows:

```
MAJORITYMV(A)
1 majority_index = 0
2 count = 1
3 for i in 1 to len(A)
4       if a[majority_index] == a[i]
5               count = count + 1
6       else
7               count = count -1
8       if count == 0
9               majority_index = i
10              count = count + 1
11 return a[majority_index]
```

The above procedure will run in O(n) time because it will iterate through the array only once. It also doesn't require any additional space to store the count of the elements.

Observations:
The majority element problem though contains optimal substructure; it cannot be solved through a purely dynamic programming approach or a greedy approach. To conclude, there are problems which even though exhibit optimal substructure, cannot be solved in a purely dynamic programming or greedy approach.

Travelling Salesman Problem

It is attached as a separate file named: TravellingSalesmanProblem_Gurleen_Sudhanshu.pdf

Reference:

1. Introduction to Algorithms 3rd edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
2. Dynamic Programming, Wikipedia Article - https://en.wikipedia.org/wiki/Dynamic_programming
3. Greedy Algorithm, Wikipedia Article - https://en.wikipedia.org/wiki/Greedy_algorithm
4. Patience Sorting, Wikipedia Article - https://en.wikipedia.org/wiki/Patience_sorting
5. Longest Increasing Subsequences: From Patience Sorting to the Baik-Deift-Johansson Theorem, David Aldous|University of California, Berkeley and Persi Diaconis|Stanford University, May 17, 1999
6. Boyer–Moore Majority Vote Algorithm, Wikipedia Article - https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_majority_vote_algorithm